Python for Data processing

# Lecture 6:
# EDA, rules of thumb and big picture

Gleb Ivashkevich

# What we already know

- Numpy
- PyTorch
- pandas
- some plotting

# Today

- **exploratory data analysis**
- **rules of thumb** and common mistakes
- **big picture** of DS and ML

# Exploratory data analysis

# Origin

It all starts with **questions**.

Not about data, but **about real world**.

**Why** it works like this?

**Can we explain** why something happens?

**Can we predict** X?

**Can we reinvent** our product with data?

# Why DS and ML

Two reasons:

- **create** something new
- **improve** something existing

# Questions and answers

When answer the questions are look at data

Do we **have** the data needed?

Is quality of this data **good enough**?

**Can we process** this data?

**Can we answer** the questions with this data?

# It’s iterative

You start answer questions, and you discover **new questions** worth asking

Target may **shift**

Questions may turn out to be **trivial**

You may **hit a wall**

That’s ok.

# Walls

Sometimes it's not possible to either answer the questions you have, or ask new ones: **data is too weak.**

**Find** new one, or **drop** it.

# Not just questions

We do not want to just know something new about the world outside.

We want to have **actionable insights**.

And because they are actionable, it's your responsibility to provide **deep** and **accurate** insights.

# Exploring the data

**Goals:**

- assess data **quality**
- understand data **structure**
- get basic (or complex) **insights**
- plan **modeling**
- plan **presentation** of your results
- plan **integration**

# Data quality

**Problem: data is usually quite bad**

- missing values
- errors
- signal may be not there
- not enough data

# Data structure

**Problem:**

- **types** and **meaning** of variables
- **ranges**
- **statistics** (histograms, counts)
- internal **relationships**
- potential **derived features**
- potential **external/additional data sources**

# Insights

**You may discover:**

- **tricky facts** about the world
- **potential problems** in reality on the ground
- sources of **improvement**
- new ways of doing things

# Presenting

**Visualizations matter**

- help you to understand data
- ~~help you to~~ **communicate your results**

But they only matter, if they are **clear enough**

# **Presenting:** mistakes

**Presenting with notebooks:**

- stakeholders may be **overwhelmed**
- **notebooks are fluid**, your “report” may be gone very soon

**Remedies:**

- plain old **slides**: concise and short
- Viola, Bokeh, Dash, etc.

# **Presenting:** mistakes

**Visualizations:**

- visualizations are **not "readable"**
- over-visualization

**Remedies:**

- try to stick to **classical** visualizations (line/scatter/bar/pie)
- if there's no choice, consider simple **interactive dashboard**

# **Presenting:** mistakes

**Context:**

- not setting the **stage**
- reporting **process**, not **results**

**Remedies:**

- explain the **goal**
- support your **approach**, describe process **shortly**
- focus on **results**(both + and -) and **next steps**

# Best and worst practices

# Code quality

Code quality **matters**: we're doing ML, but technically it's still **software development.**

**Low** code quality:

- bugs,
- delayed deployment,
- unneeded iterations,
- sub-optimal performance.

# Code quality

High code quality:

- read **PEP8**(or similar style guide for your language of choice)
- use linter,
- prefer **readability** and **transparency**,
- **structure**, but **not over-structure**.

# Reproducibility

You results **must** be reproducible:

- same computation must produce same results,
- **plan** experiments,
- **log** experiments,
- create **artefacts**,
- **split configuration** and **parameters** from code,
- set **random seeds**.

# Versioning

**No version control = no reproducibility.** Period.

Code versioning:

- nothing is lost,
- one experiment = one commit,
- streamline deployment.

**Git.**

# Versioning

**No version control = no reproducibility.** Period.

**Artefacts**(models, features, etc.) and **pipelines** versioning:

- experiments can be reproduced,
- experiments can be compared,
- streamline deployment.

**DVC, Kedro, MLFlow.**

# Project structure

**Separate:**

- code from configuration and parameters,
- code and config from data,
- generally useful utilities from exploratory and training code.

**Benefits:**

- easily to extend later on,
- streamline deployment.

# Black boxing

**Main and most severe** ML sin:

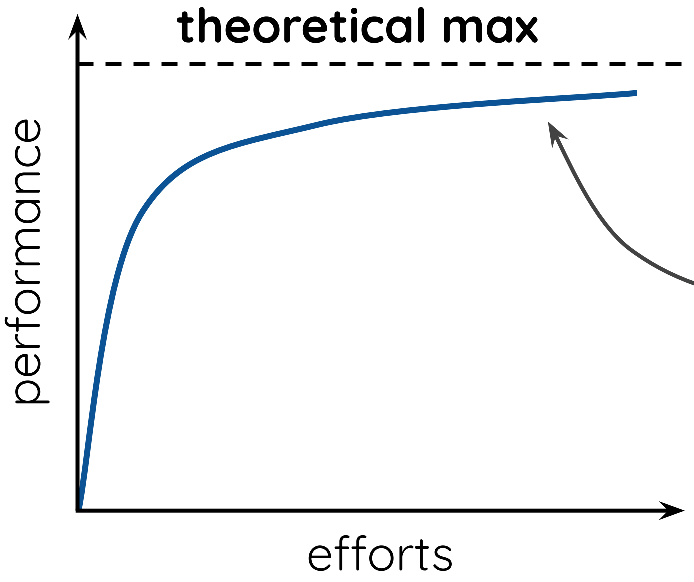
- throwing data into a model without understanding,
- throwing data into a model without rationale,
- not trying simple models first.

**Consequences:**

- actual performance hard to put into context,
- various deployment-time surprises.

# Black boxing

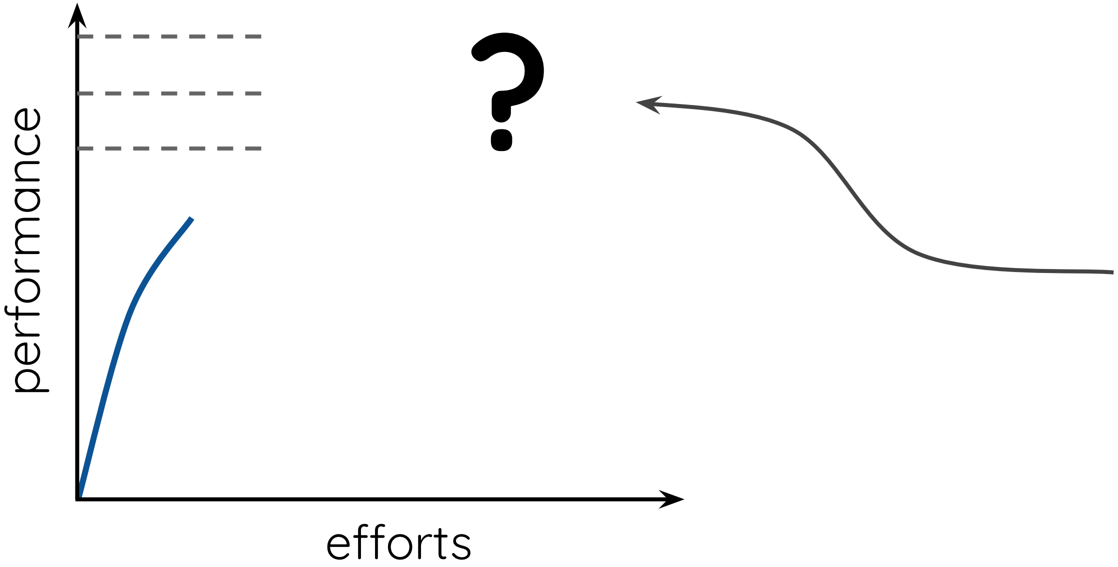
**Diminishing returns:**




You want to jump **here** with the best and most advanced model

# Black boxing

**Diminishing returns:**




In reality, you
jump **here**

# Baselines

**Instead of jumping into the most advanced model:**

- establish robust baseline,
- try to preserve interpretability,
- move incrementally (this has nothing to do with speed)

**Benefits:**

- progress is quantifiable,
- less surprises,
- more trust.

# **Big picture:**
# Python ecosystem

# **Combine tools** to solve large problems

**Steps** to build something:

- get data
- explore
- model
- present
- deploy
- iterate (usually in explore - model - present cycle)

# Slow and fast data

**Slow data** is sitting in DBs and is updated from time to time

- dump, queues

**Fast data** is hitting your backend systems at a very high rate and must be processed quickly

- streaming processing or alike

# Get data

**From SQL DB:**

- sqlalchemy

**Web:**

- requests

**From other storage systems:**

- specific APIs and packages

# Get data

**To process it immediately/quickly:**

- Queues
- Dask/Ray/Faust
- Spark/Storm/Kafka

# Explore

**Structured data:**

- pandas

**Images:**

- OpenCV, **skimage**

**Use:**

- notebooks (**tqdm** is useful)
- visualizations

# Model

**For structured data:**

- **sklearn** estimators
- XGBoost, CatBoost, LightGBM

**For images and other unstructured data:**

- PyTorch, TensorFlow/Keras

**Distributed:**

- Horovod (from Uber), Dask, Ray

# Present

**Visualizations** matter:

- Matplotlib, Seaborn, Bokeh, Plotly

**Dashboards** may help:

- Bokeh, Dash, Grafana

Viola, reveal.js instead of PDF’s

# Deploy

For **classical** models:

- RESTful API with **falcon** or **flask**

For **deep learning** models:

- GraphPipe
- PyML
- TensorFlow serving

**Big picture:**

Data, it's all about data

# Data is different now

Data from **IoT** devices:

- streaming
- columnar
- graph

And **more** to come:

- edge computing
- distributed computing

# Columnar databases

Data may be inherently (time) **ordered**:

- row storage is **inefficient**
- traditional databases are really **bad** in analytic workloads

**Columnar engines** and databases to the rescue:

- PostgreSQL + cstore_fdw
- ClickHouse (Yandex)

# Columnar formats

Apache **Arrow**

Apache **Parquet**

# When data is huge

Data may still be either too large, or coming to fast:

**Hadoop stack**

**It's Java**

**But there's Scala**

# When data is huge

**Apache Spark:** distributed analytics engine

- in memory
- can handle streaming jobs
- knows about ML
- and graph data
- and even TensorFlow!

# When data is huge

Native way to use Spark is with **Scala**

Scala may look a bit crazy at first, but it’s **powerful and flexible**

**Saves a lot of time** compared to Java

# When data is huge

**Scala:**

- functional or object-oriented
- strong typing
- but with type inference
- works on JVM
- interoperate with Java

# Compute faster

# Julia

But why?

- C/C++ is costly in development, but fast at runtime
- Python is cheap, but is slow at runtime
- Python has too many layers of abstraction

Julia promises to be **the best of two worlds**

# Julia

**Features:**

- Julia is **fast**
- **multiple dispatch**
- **parallel and distributed** computing
- calls to C functions are **native**
- calls to Python are **simple**
- great support of **GPU computing**

# Oldie, but goodie

# R

Robust and well respected tool for statistical computing

- **long history**
- **great community**
- **problems** with integration
- **non-uniform interfaces**

# Wrap-up

# Next

**New hardware** is coming and **IoT** is on the rise

**New ways to compute: edge** and **distributed**

**Quantum computing?**

**Decline** of 1-st gen deep learning?

**Decline** of Python?

**AI nationalism**

# Takeaway note

Rely on **fundamentals**

Keep an eye on **modern developments**

**Adapt**, as only few things remain constant:

- **probability** theory,
- **first principles** approach,
- general engineering **craftsmanship**.

# Takeaway note

Have fun in this fascinating journey :)

questions?